Encoding the Factorisation Calculus
(Representing the Intensional in the Extensional)

Reuben N. S. Rowe
Department of Computer Science
University College London
r.rowe@ucl.ac.uk

Jay and Given-Wilson have recently introduced the Factorisation (or SF-) calculus as a minimal fundamental model of *intensional* computation. It is a combinatory calculus containing a special combinator, F, which is able to examine the internal structure of its first argument. The calculus is significant in that as well as being combinatorially complete it also exhibits the property of structural completeness, i.e. it is able to represent any function on terms definable using pattern matching on arbitrary normal forms. In particular, it admits a term that can decide the structural equality of any two arbitrary normal forms.

Since SF-calculus is combinatorially complete, it is clearly at least as powerful as the more familiar and paradigmatic Turing-powerful computational models of λ-calculus and Combinatory Logic. Its relationship to these models in the converse direction is less obvious, however. Jay and Given-Wilson have suggested that SF-calculus is strictly more powerful than the aforementioned models, but a detailed study of the connections between these models is yet to be undertaken.

This paper begins to bridge that gap by presenting a faithful encoding of the Factorisation Calculus into the λ-calculus preserving both reduction and strong normalisation. The existence of such an encoding is a new result. It also suggests that there is, in some sense, an equivalence between the former model and the latter. We discuss to what extent our result constitutes an equivalence by considering it in the context of some previously defined frameworks for comparing computational power and expressiveness.

1 Introduction

Mathematical models of computation are useful in studying the formal properties of programming practice. Indeed, the field of computing today arose partly out of the study of such abstract models: namely Turing Machines [33], the λ-calculus [8], and Combinatory Logic [10], which are consequently considered to be archetypal computational models. It is standard practice to qualify the abilities, or expressiveness, of a formal model of computation by demonstrating that it may *simulate* (and be simulated by) the operation of other formal models. This is the very essence of the notion of *Turing-completeness*, which encapsulates the intuition that a model may carry out any operation that is 'effectively computable'. To construct such a simulation one must first give an injective mapping, showing how the terms of the source model may be represented by terms of the target. For example, this is the basis behind the process of Gödelization and the Church encoding of natural numbers [22]. Two basic properties are then required: that each atomic operational step of the source model is reflected by one or more steps of the target, and that a program of the target model *terminates* whenever the corresponding source program does. The former is a key ingredient of Landin's influential work on comparing languages [25], while the latter is used as a criterion for comparing expressiveness by, e.g., Felleisen [12]. Formal definitions of encodings incorporating these properties, referred to as "faithful", are already in use by the 90s, e.g. in [2], and are now common (see e.g. [15]).

S. Crafa and D. Gebler (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2015)
EPTCS 190, 2015, pp. 76–90, doi:10.4204/EPTCS.190.6

Recently, Jay's work on formal models of generic pattern matching [17] have led, in collaboration with Given-Wilson, to the formulation of the *Factorisation Calculus*. This is a combinatory calculus comprising two combinators: the S combinator, familiar from Combinatory Logic; and a new F combinator. The purpose of the latter is to enable arbitrary (head) normal terms to be *factorised*, that is split into their constituent parts, thereby allowing the examination of the internal structure of terms. This endows Factorisation Calculus with an interesting and powerful property: that of *structure completeness*. This means that any function on terms themselves definable by pattern matching over *arbitrary* normal forms is *representable*. Thus, Factorisation Calculus can be viewed as a minimal, fundamental model characterising not only the abstract notion of *pattern matching* but also *intensional computation*.

Jay and Given-Wilson show that Factorisation Calculus is Turing-complete, demonstrating a straightforward simulation of Combinatory Logic in their calculus. Moreover, due to its structural completeness, there is a term of Factorisation Calculus which can decide the structural equality of any two arbitrary normal forms. Conversely, factorisation and structural equality of normal forms *cannot* be so represented in λ-calculus and Combinatory Logic, thus these models are *not* structure complete. This hints at some sort of disparity in the expressivity of the two models. In their original and subsequent research [18, 14, 19], Jay and Given-Wilson speculate that the added expressive power may manifest itself in a non-existence result for simulations of Factorisation Calculus in λ-calculus, but this is not pursued in detail.

We show that there *does* exist a simulation of Factorisation Calculus within λ-calculus. The existence of such a simulation has not been demonstrated before, and this is the primary contribution of our paper. The simulation is made possible by a construction due to Berarducci and Böhm, which shows how to encode a certain class of term rewriting systems within λ-calculus. We show that Factorisation Calculus can be simulated by such a term rewriting system, whence the result follows. In the classical framework, our result signifies that Factorisation Calculus is no more powerful than λ-calculus. Thus there appears to be a mismatch between our result and the structure completeness property that the standard simulation-based notion of equivalence does not account for. To begin to try and resolve this, we consider some research in the literature which refines the concept of computational equivalence and discuss how our result relates to this.

Outline The rest of this paper is organised as follows. Section 2 recalls Jay and Given-Wilson's Factorisation Calculus and its basic properties. Section 3 describes Berarducci and Böhm's construction for encoding so-called canonical rewrite systems within the λ-calculus. In Section 4 we present our technical contribution: a simulation of the Factorisation Calculus in the λ-calculus via this construction. Section 5 then discusses, in light of our results, how the relative expressiveness of Factorisation Calculus and λ-calculus may be characterised. Section 6 concludes and remarks on areas for future work.

2 Factorisation Calculus

We begin by presenting Jay and Given-Wilson's Factoriation Calculus itself, and review its principal properties. Factorisation Calculus, or more accurately SF-calculus[1], is a combinatory calculus whose terms are those of the free algebra over the two-element signature containing the combinators S and F, which each reduce upon being applied to three arguments. The former is the familiar combinator from Combinatory Logic [10] which applies its first and second arguments to duplicates of its third. The F combinator, on the other hand, introduces new capabilities in the form of *factorisation*: it is

[1] We may say that any combinatory calculus that is structure complete is *a* factorisation calculus.

able to examine the *internal* structure of its first argument and process its second and third in different ways depending on whether that argument is *atomic* (i.e. itself a combinator) or *compound* (i.e. a partial application). To illustrate this, consider how the F combinator reduces in the following two instances:

$$\mathsf{F}\,\mathsf{S}\,M\,N \to M \qquad\qquad \mathsf{F}\,(\mathsf{S}\,X)\,M\,N \to N\,\mathsf{S}\,X$$

Observe that when the first argument is atomic, it eliminates its third argument and returns its second. On the other hand, when the first argument is compound it eliminates its second argument and *factorises* the first into its left- and right-hand constituent components, passing these *separately* to its third argument.

Formally, SF-calculus is defined as follows.

Definition 2.1 (SF-calculus [18, § 4])**.** *The SF-calculus is a combinatory rewrite system over terms (ranged over by uppercase roman letters M, N, etc.) given by the following grammar:*

$$M, N ::= \mathsf{S} \mid \mathsf{F} \mid MN$$

Terms of the form S, F, $\mathsf{S}M$, $\mathsf{F}M$, $\mathsf{S}MN$, *or* $\mathsf{F}MN$ *(i.e. partially applied combinators) are called* factorable forms*. Reduction of terms is the smallest contextually closed binary relation $\to_{SF}$ on terms (with the reflexive transitive closure denoted by $\to^*_{SF}$) satisfying:*

$$\begin{aligned}
\mathsf{S}\,M\,N\,X &\to_{SF} M\,X\,(N\,X) && \\
\mathsf{F}\,O\,M\,N &\to_{SF} M && \text{if } O \text{ is } \mathsf{S} \text{ or } \mathsf{F} \\
\mathsf{F}\,(P\,Q)\,M\,N &\to_{SF} N\,P\,Q && \text{if } P\,Q \text{ is a factorable form}
\end{aligned}$$

Reduction of SF-calculus is confluent, and the K combinator of Combinatory Logic can be represented in SF-calculus by $\mathsf{F}\,\mathsf{F}$ (also, indeed, by $\mathsf{F}\,\mathsf{S}$). Thus, there is a trivial encoding of Combinatory Logic in SF-calculus which preserves reduction and strong normalisation [14].

The behaviour of the F combinator gives SF-calculus an *intensional* quality: one may define higher order functions in SF-calculus which discriminate between functions whose *implementations* are different even when those functions are *extensionally* equal (i.e. produce identical outputs for identical inputs). For example, for any normal form X, the term $I_X \equiv \mathsf{S}\,(\mathsf{F}\,\mathsf{F})\,X$ implements the identity function (i.e. $I_X\,M \to^*_{\mathsf{SF}} M$ for all M) and thus all such terms are extensionally equal. However, an SF-term T can be constructed which distinguishes them (by behaving as $T\,I_X \to^*_{\mathsf{SF}} X$). Moreover, one can construct an SF-term that can decide the equality of any two *arbitrary* normal forms.

The intensional behaviour of SF-calculus is formally characterised by a property called *structure completeness*, which captures the notion that every *symbolic computation* (i.e. Turing-computable symbolic function) on normal forms is represented by some term.

Definition 2.2 (Structure Completeness [18, § 7-8])**.** *Let $\mathscr{C}$ be a confluent combinatory calculus whose terms include variables, with reduction relation $\to^*_{\mathscr{C}}$. Define* patterns *to be the* linear *normal forms (i.e. containing no more than one occurrence of each variable), and* matchable forms *to be partially applied combinators.*

1. *A* match $\{U/P\}$ *of a pattern P against a term U may be defined to* succeed *with a substitution of terms for variables, or* fail *as follows (where* Id *denotes the identity function and $\uplus$ the disjoint union of substitutions with match failure as an absorbing element):*

$$\begin{aligned}
\{U/x\} = [U/x] \qquad \{A/A\} = \mathsf{Id} && \text{(if } A \text{ atomic)} \\
\{UV/PQ\} = \{U/P\} \uplus \{V/Q\} && \text{(if } UV \text{ a compound)} \\
\{U/P\} = \mathsf{fail} && \text{(otherwise, if } U \text{ matchable)}
\end{aligned}$$

2. *A* case *is an equation of the form $P = M$ where P is a pattern and M an arbitrary term, which defines a* symbolic function $\mathscr{G}$ *on terms by $\mathscr{G}(U) = \sigma(M)$ if $\{U/P\}$ succeeds with substitution σ, and $\mathscr{G}(U) = U$ if it fails.*
3. *A confluent combinatory calculus is* structure complete *if for every pattern P and term M, there is some term G such that $GU \to^*_{\mathscr{C}} \mathscr{G}(U)$ for every term U on which $\mathscr{G}$ is defined; i.e. the symbolic function defined by every case is represented by some term.*

Structure completeness subsumes combinatorial completeness since $\lambda x.M$ is given by the case $x = M$.

Theorem 2.3 ([18, Cor. 8.4])**.** *SF-calculus is structure complete.*

The F combinator itself represents a symbolic computation $\mathscr{F}$, namely that of factorisation:

$$\begin{aligned} \mathscr{F}(A, M, N) &= M && \text{if } A \text{ is atomic} \\ \mathscr{F}(PQ, M, N) &= NPQ && \text{if } PQ \text{ is compound} \end{aligned}$$

A significant (and arguably remarkable) fact is that $\mathscr{F}$ cannot be represented in Combinatory Logic (for definitions of atomic and compound appropriate thereto).

Theorem 2.4 ([18, Thm. 3.2])**.** *Factorisation of SK-combinators is a symbolic computation that is not representable in Combinatory Logic.*

The equality predicate on normal forms also has no representation in Combinatory Logic. Thus, there exist (symbolic) functions, which are clearly 'computable' from an empirical point of view, that are not (directly) representable in Combinatory Logic (there is also a similar result for λ-calculus [4]). This result clearly points towards some form of added expressivity possessed by SF-calculus over the archetypal computational models. It is to this issue that we will return in Section 5.

3 Strongly Normalising Solutions of Equational Systems in λ-calculus

We now reiterate the interpretation result of Berarducci and Böhm [5], upon which our technical contribution rests. Essentially, this result says that systems of equations for a particular class of term algebras can be given solutions in the λ-calculus such that the representation of each atomic term of the algebra is *strongly normalising* thus having a *normal form*. Moreover, when the set of equations is interpreted as a rewrite system the encoding of terms preserves reduction and strong normalisation.

We assume the usual definitions of the λ-calculus without further explanation (readers may refer to [4] for details), with Λ denoting the set of lambda terms, $\to^*_\beta$ denoting the (multi-step) β-reduction relation, and $=_\beta$ denoting β-equality (i.e. the equivalence relation on lambda terms induced by β-reduction). Furthermore, we also assume the familiar algebraic notion of the set $\mathrm{Ter}(\Sigma)$ of (Σ-)*terms* over the signature (set of *function symbols*, each with an associated arity) Σ. We can then also consider the set $\Lambda(\Sigma)$ of *extended* lambda terms (i.e. lambda terms which may contain Σ-terms); notice that both $\mathrm{Ter}(\Sigma)$ and Λ are (strict) subsets of $\Lambda(\Sigma)$.

Definition 3.1 (Canonical Systems of Equations)**.** *Fix a signature Σ and let $\mathscr{E}$ be a set of equations between terms $t \in \mathrm{Ter}(\Sigma)$. We say that $\mathscr{E}$ is* canonical *if Σ can be partitioned into two disjoint subsets Σ_0 and Σ_1 (i.e. $\Sigma = \Sigma_0 \cup \Sigma_1$) such that: each equation in $\mathscr{E}$ is of the form $f(c(x_1,\ldots,x_m),y_1,\ldots,y_n) = t$ with $c \in \Sigma_0$ and $f \in \Sigma_1$ and where the variables $x_1,\ldots,x_m,y_1,\ldots,y_n$ are all distinct and form a superset of the variables in the term t; and for each distinct pair $(c,f) \in \Sigma_0 \times \Sigma_1$ there is at most one equation in $\mathscr{E}$ of this form. We say that $\mathscr{E}$ is* complete *if for each distinct pair $(c,f) \in \Sigma_0 \times \Sigma_1$ there is exactly one such equation in $\mathscr{E}$.*

Canonical systems of equations, then, partition the signature into a set Σ_0 of (algebraic datatype) *constructors*, and Σ_1 of *programs* defined by pattern matching over the constructors on the first argument. Notice that any incomplete canonical system of equations can trivially be made complete by adding equations for the missing cases which simply project one of the function's arguments[2].

As an example of a canonical system of equations, we may observe that the usual recursive definition of addition over the datatype of (Peano) natural numbers is such a system:

$$\mathsf{add}\,(\mathsf{zero},x) = x \qquad \mathsf{add}\,(\mathsf{succ}\,(x),y) = \mathsf{succ}\,(\mathsf{add}\,(x,y))$$

We have a signature containing one function symbol add, one nullary constructor zero, and one unary constructor succ; moreover in this simple case, the equation system is already complete. In fact, every partial recursive function (on natural numbers) can be defined by a canonical system of equations [5, 6].

Given an equational system $\mathscr{E}$ over a signature Σ, we can also take it to define a term rewriting system on $\mathrm{Ter}(\Sigma)$ by reading each equation as a *rewrite* rule, i.e. $f_i(c_j(x_1,\ldots,x_m),y_1,\ldots,y_n) \to t$. We will write $\to_{\mathscr{E}}$ for the (one-step) reduction relation of the rewrite system defined by $\mathscr{E}$ in this way (i.e. the smallest binary relation on terms satisfying the rewrite rules and closed under substitution and contexts), and $\to^*_{\mathscr{E}}$ for its reflexive, transitive closure (i.e. multi-step reduction). Ultimately, the aim is to interpret equational systems (and their associated rewrite systems) within λ-calculus.

Definition 3.2 (Interpretations)**.** *A* representation *of the signature Σ is a function $\phi : \Sigma \to \Lambda$ from the function symbols of Σ to (closed) lambda terms, and induces a map $(\cdot)^\phi : \Lambda(\Sigma) \to \Lambda$ in the obvious way, namely by $x^\phi = x$, $(\lambda x.M)^\phi = \lambda x.M^\phi$, $(MN)^\phi = M^\phi N^\phi$, and for $f \in \Sigma$, $f\,(t_1,\ldots,t_n)^\phi = \phi(f)\,t_1{}^\phi \ldots t_n{}^\phi$. We say that a representation ϕ* satisfies *(or* solves*) $\mathscr{E}$ if for each equation $t_1 = t_2$ (and corresponding rewrite rule $t_1 \to t_2$) in $\mathscr{E}$ we have $t_1{}^\phi =_\beta t_2{}^\phi$ (and correspondingly also $t_1{}^\phi \to^*_\beta t_2{}^\phi$). When a representation ϕ satisfies $\mathscr{E}$, we say that ϕ is an* interpretation *(or a* solution*) of $\mathscr{E}$ within λ-calculus.*

The following construction gives a special kind of representation for canonical systems of equations.

Definition 3.3 (Canonical Representations)**.** *Let $\mathscr{E}$ be a canonical system of equations that partitions the signature Σ into constructors $\Sigma_0 = \{c_1,\ldots,c_r\}$ and programs $\Sigma_1 = \{f_1,\ldots,f_k\}$. Without loss of generality we may assume that $\mathscr{E}$ is complete, and so for each $1 \le i \le k$ and $1 \le j \le r$ let $b_{(i,j)}$ denote the term t such that $f_i(c_j(x_1,\ldots,x_m),y_1,\ldots,y_n) = t \in \mathscr{E}$.*

We will make use of the following notational abbreviations:

- *Let $\langle t_1,\ldots,t_n\rangle$ denote the Church n-tuple, i.e. $\lambda x.xt_1\ldots t_n$.*
- *Let Π^n_k (where $1 \le k \le n$) be the n-ary k^{th} projection function, i.e. $\lambda x_1\ldots x_n.x_k$.*
- *For $k \ge i > 1$, let $t_i,\ldots,t_k,\ldots,t_{i-1}$ denote the cyclic permutation of $t_1,t_2,\ldots,t_k$ beginning with t_i; (in an abuse of notation we may also take $t_i,\ldots,t_k,\ldots,t_{i-1} = t_1,t_2,\ldots,t_k$ when $i = 1$).*

We now define two disjoint representations ϑ and ζ for constructors and programs respectively.

(Representation of Constructors) *For each $1 \le i \le r$, we define the representation of the constructor c_i as follows (where n is the arity of c_i):*

$$\vartheta(c_i) = \lambda x_1\ldots x_n f.\, f\Pi^r_i x_1\ldots x_n f$$

(Representation of Programs) *We choose k distinct fresh variables $v_1,\ldots,v_k$ not occurring in $\mathscr{E}$ and fix a 'pre-representation', ψ, of Σ_1 defined by $\psi(f_i) = \langle v_i,\ldots,v_k,\ldots,v_{i-1}\rangle$ for each $1 \le i \le k$.*

[2] Alternatively, one might want to introduce a new nullary constructor (denoting an 'error' value) and add equations for the missing cases that simply return this value.

Using this representation, and the representation of constructors defined above, we then define $k \times r$ lambda terms $t_{(i,j)}$ ($1 \leq i \leq k$, $1 \leq j \leq r$), using the equations in $\mathscr{E}$ as follows:

$$t_{(i,j)} = \lambda x_1 \ldots x_m v_i \ldots v_k \ldots v_{i-1} y_1 \ldots y_n . (b_{(i,j)}{}^{\psi})^{\vartheta}$$

where $f_i(c_j(x_1,\ldots,x_m),y_1,\ldots,y_n) = b_{(i,j)} \in \mathscr{E}$ is the equation defining the behaviour of f_i when given a datum constructed using c_j as its first argument. We now define k terms, each one a Church r-tuple collating the bodies of all the cases for one of the programs in Σ_1, as follows:

$$t_i = \langle t_{(i,1)}, \ldots, t_{(i,r)} \rangle$$

(where $1 \leq i \leq k$). Each program is then represented by a Church k-tuple containing the collated representations of each program defintion, beginning with its own. That is, ζ is defined by:

$$\zeta(f_i) = \langle t_i, \ldots, t_k, \ldots, t_{i-1} \rangle \quad (1 \leq i \leq k)$$

The representation $\phi = \vartheta \cup \zeta$ is called a canonical *representation of Σ with respect to $\mathscr{E}$.*

To gain some insight into the construction defined above, one can observe that it is related to an encoding of data attributed to Scott[3] (and thus commonly referred to in the literature as the Scott encoding), which has subsequently been developed by others (e.g. [30, 27, 16, 31]). In the more familiar 'standard' encoding of functions, a fixed-point combinator is used to solve any recursion in the definition. This has the effect of making recursion *explicit*, and thus the representations of recursive functions have infinite expansions consisting of a 'list' of distinct instances of the function body, one for each recursive call that may be made. Applying the function to a datum then corresponds to a *fold* of the datum over this list, which discards the remaining infinity of recursive calls once the base case is reached. Therefore, as described by Böhm et al. [6, §3], in this scheme functions are 'diverging objects which, when applied to data, may "incidentally" converge'. In encodings of the Scott variety, the recursive nature of functions is kept *implicit* and, while still triggered by application to a datum, only reproduced 'on demand'. Hence we obtain finite objects which now 'may "incidentally" diverge' when applied to data[4].

To explicate the particular encoding specified by Definition 3.3, we point out that the representation of a constructor is a (lambda) function that takes in the appropriate number of arguments (the *sub-data* of the datum that is subsequently constructed) and then waits to be given a function, which will be the program to be executed. Now, looking at how the constructor representation uses this function argument, we see that programs should expect to be given a projection function, followed by a number of sub-data, and then they are also passed *a copy of themselves*. It is this final element which is the key to Scott-type encodings, and allows recursion to be kept implicit. Looking now at the representation of programs we see that they are Church *k*-tuples containing an element for each program defined by $\mathscr{E}$ (each of which is a Church *r*-tuple, where each element is a representation of one of the cases of that program's definition). Thus the representation of each program contains the definition of *every* program defined by $\mathscr{E}$; in particular it will contain the definition of each program which it may itself invoke. To illustrate in more detail how the encoding works, we can consider the general reduction sequence of a term representing the application of some program prog_i to some arguments, the first of which is a datum constructed as $c_j(d_1,\ldots,d_m)$:

$$(\mathsf{prog}_i\,(c_j\,(d_1,\ldots,d_m))\,\mathsf{arg}_1 \ldots \mathsf{arg}_m)^{\phi} = \langle t_i,\ldots,t_{i-1}\rangle\,(c_j\,(d_1,\ldots,d_m))^{\phi}\,\mathsf{arg}_1^{\phi} \ldots \mathsf{arg}_m^{\phi}$$

[3] The citation can be found in Curry, Hindley and Seldin [11, p. 504].

[4] This reversed form of the slogan is also due to Böhm et al., and illustrates the dual nature of the Scott and Church encodings.

$$\to^*_\beta \quad (\lambda x.x\,t_i \ldots t_{i-1})\,(\lambda f.f\,\Pi^r_j d_1 \ldots d_m f)\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m \tag{1}$$

$$\to_\beta \quad (\lambda f.f\,\Pi^r_j d_1 \ldots d_m f)\,t_i \ldots t_{i-1}\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m \tag{2}$$

$$\to_\beta \quad t_i\,\Pi^r_j d_1 \ldots d_m\,t_i\,t_{i+1} \ldots t_{i-1}\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m \tag{3}$$

$$= \quad \langle t_{i,1}, \ldots, t_{i,r} \rangle\,\Pi^r_j d_1 \ldots d_m\,t_i\,t_{i+1} \ldots t_{i-1}\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m \tag{4}$$

$$= \quad (\lambda x.x\,t_{i,1} \ldots t_{i,r})\,\Pi^r_j d_1 \ldots d_m\,t_i\,t_{i+1} \ldots t_{i-1}\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m \tag{5}$$

$$\to_\beta \quad \Pi^r_j\,t_{i,1} \ldots t_{i,r}\,d_1 \ldots d_m\,t_i\,t_{i+1} \ldots t_{i-1}\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m \tag{6}$$

$$\to_\beta \quad t_{i,j}\,d_1 \ldots d_m\,t_i\,t_{i+1} \ldots t_{i-1}\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m \tag{7}$$

$$= \quad (\lambda x_1 \ldots x_m v_i \ldots v_k \ldots v_{i-1} y_1 \ldots y_n.(b_{(i,j)}{}^\psi)^\vartheta)\,d_1 \ldots d_m\,t_i\,t_{i+1} \ldots t_{i-1}\,\mathsf{arg}^\phi_1 \ldots \mathsf{arg}^\phi_m$$

$$\to^*_\beta \quad (b_{(i,j)}\,[d_1/x_1, \ldots, d_m/x_m, \mathsf{arg}_1/y_1, \ldots, \mathsf{arg}_n/y_n])^\phi \tag{8}$$

When the program is applied to a datum (Eq. (1)), its representation arranges to apply the datum first to the representations of each program beginning with its own, and then to the remainder of the arguments (Eq. (2)). Then, the particular structure of the datum will reduce the expression to pick out the appropriate case of the program definition to be executed, and apply it to the sub-data and the representations of each program, having duplicated the program being executed (Eqs. (3) to (7)). This then reduces to the representation of the appropriate substitution instance of the function body (Eq. (8)).

The result of Berarducci and Böhm says that a canonical representation gives an interpretation that also preserves strong normalisation.

Theorem 3.4 (Interpretation Theorem [5, Thm 3.4])**.** *Let Σ be a signature and $\mathscr{E}$ a canonical set of equations for Σ; then any canonical representation ϕ for Σ with respect to $\mathscr{E}$ is an interpretation of $\mathscr{E}$ within λ-calculus. In addition $(\cdot)^\phi$ preserves strong normalisation of closed terms.*

4 Encoding the Factorisation Calculus

In this section, we present our novel technical contribution: an encoding of SF-calculus in λ-calculus. We believe that this is the first such encoding presented in the literature. Our encoding is a faithful simulation; it preserves both the reduction behaviour of terms (thus also β-equality) and their termination behaviour (i.e. strong normalisation). In this section we shall make use of standard notation and results for term rewriting systems, details of which may be found in [24].

The key step to the encoding is to define a rewrite system behaviourally equivalent to SF-calculus that is also canonical, in the sense of Definition 3.1. It is then simply a matter of applying the construction of Berarducci and Böhm to obtain the encoding. Thus it is our translation of SF-calculus into this intermediate rewrite system that is the primary novelty of our contribution.

We are aiming to derive a set of rewrite rules that is *canonical* and so we must translate the schematic definition of Jay and Given-Wilson, as presented in Section 2, into one consisting of algebraic rewrite rules. There are two salient features of Definition 3.1 that we must take into account: that the rewrite rules must make a distinction between *programs* and *constructors*; and that the left-hand side of each rewrite rule must contain exactly one program symbol and one constructor. To obtain rewrite rules of the required form, we recast SF-calculus as a *curryfied, applicative* term rewriting system. That is, we first introduce an explicit program symbol app to denote application and use the symbols S and F solely as *constructors*. Secondly we stratify the combinators $\mathsf{C} \in \{\mathsf{S}, \mathsf{F}\}$ into sets $\{\mathsf{C}_0, \mathsf{C}_1, \mathsf{C}_2\}$ of constructors, each of which represent successive *partial* applications of their underlying combinator C. Although this

'currying' process is well-known from the world of functional programming, the reader may refer to [20, 3] for a formal definition of this process in the context of general term rewriting.

We may take the rewrite rules for the S combinator directly from the standard curryfied applicative formulation of Combinatory Logic (see e.g. [3]):

$$\mathsf{app}(\mathsf{S}_0, x) \to \mathsf{S}_1(x) \quad \mathsf{app}(\mathsf{S}_1(x), y) \to \mathsf{S}_2(x, y) \quad \mathsf{app}(\mathsf{S}_2(x, y), z) \to \mathsf{app}(\mathsf{app}(x, z), \mathsf{app}(y, z))$$

The rules for producing the partial applications of the F combinator are similarly straightforward:

$$\mathsf{app}(\mathsf{F}_0, x) \to \mathsf{F}_1(x) \qquad \mathsf{app}(\mathsf{F}_1(x), y) \to \mathsf{F}_2(x, y)$$

The rewrite rule for the full application of the F combinator is more tricky because we must find a way of implementing its two possible reductions. As in the original formulation of SF-calculus, since the choice of which reduction to make is determined by the structure of the first argument we should like to be able to use the pattern-matching capabilities inherent in the term rewriting discipline, e.g. by giving rewrite rules such as:

$$\mathsf{app}(\mathsf{F}_2(\mathsf{S}_0, y), z) \to y \qquad \mathsf{app}(\mathsf{F}_2(\mathsf{F}_1(x), y), z) \to \mathsf{app}(\mathsf{app}(z, \mathsf{F}_0), x)$$

However these rules are *not* canonical since they contain two occurrences of a constructor: they are pattern-matching 'too deeply'. We can circumvent this by introducing an auxiliary *program* symbol $\mathsf{f\text{-}reduce}$ and then having the rewrite rule for the F_2 case of app delegate to this new program:

$$\mathsf{app}(\mathsf{F}_2(x, y), z) \to \mathsf{f\text{-}reduce}(x, y, z)$$

Since $\mathsf{f\text{-}reduce}$ is an independent program symbol, and only needs to pattern match on its first argument to determine which result to compute, we may give canonical rewrite rules for it, such as the following:

$$\mathsf{f\text{-}reduce}(\mathsf{S}_0, y, z) \to y \qquad \mathsf{f\text{-}reduce}(\mathsf{F}_1(x), y, z) \to \mathsf{app}(\mathsf{app}(z, \mathsf{F}_0), x)$$

We now have all the components to be able to present a canonical rewrite system that faithfully implements SF-calculus.

Definition 4.1 (Currified Applicative SF-Calculus)**.** *Let $\Sigma_{SF} = \Sigma_0 \cup \Sigma_1$ be the signature comprising the set $\Sigma_0 = \{\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2, \mathsf{F}_0, \mathsf{F}_1, \mathsf{F}_2\}$ of constructors and the set $\Sigma_1 = \{\mathsf{app}, \mathsf{f\text{-}reduce}\}$ of programs.* Curryfied Applicative SF-calculus *is the term rewriting system $\mathsf{SF}^{\mathscr{C}}_{@}$ defined by the rewrite rules given in Figure 1 over the signature Σ_{SF}. We denote its one-step and many-step reduction relations by $\to_{\mathsf{SF}^{\mathscr{C}}_{@}}$ and $\to^*_{\mathsf{SF}^{\mathscr{C}}_{@}}$.*

Notice that the rewrite rules of $\mathsf{SF}^{\mathscr{C}}_{@}$ are canonical, in the sense of Definition 3.1, and that they are also *complete*. We also remark that $\mathsf{SF}^{\mathscr{C}}_{@}$ is an *orthogonal* term rewriting system [24, Def. 2.1.1].

It is interesting to observe that our implementation of SF-calculus as a canonical applicative term rewriting system has involved an application of the *Visitor* design pattern[5] [13]. When it comes to reducing a complete application of the F combinator, the computation must proceed based on the particular identity of some object (i.e. the first argument), but *without having any knowledge of that identity*. The solution is to apply the visitor pattern, which involves invoking a new 'visit' operation (that we call $\mathsf{f\text{-}reduce}$) on the object, which in response executes the appropriate behaviour based on its *self*-knowledge of its own identity. We do not think it is entirely coincidental that the visitor pattern has arisen in our work: its connection with structural matching has already been noted [28], and investigating this connection further is an avenue for future research.

There is a straightforward translation from SF-calculus to $\mathsf{SF}^{\mathscr{C}}_{@}$.

[5] In fact, the encoding that we are presenting in this paper arose as a direct result of considering how the Factorisation Calculus could be implemented in (Featherweight) Java.

$$\mathsf{app}(\mathsf{S}_0, x) \to \mathsf{S}_1(x) \qquad \mathsf{app}(\mathsf{F}_0, x) \to \mathsf{F}_1(x)$$
$$\mathsf{app}(\mathsf{S}_1(x), y) \to \mathsf{S}_2(x, y) \qquad \mathsf{app}(\mathsf{F}_1(x), y) \to \mathsf{F}_2(x, y)$$
$$\mathsf{app}(\mathsf{S}_2(x, y), z) \to \mathsf{app}(\mathsf{app}(x, z), \mathsf{app}(y, z)) \qquad \mathsf{app}(\mathsf{F}_2(x, y), z) \to \mathsf{f\text{-}reduce}(x, y, z)$$

$$\mathsf{f\text{-}reduce}(\mathsf{S}_0, y, z) \to y \qquad \mathsf{f\text{-}reduce}(\mathsf{S}_1(x), y, z) \to \mathsf{app}(\mathsf{app}(z, \mathsf{S}_0), x)$$
$$\mathsf{f\text{-}reduce}(\mathsf{F}_0, y, z) \to y \qquad \mathsf{f\text{-}reduce}(\mathsf{F}_1(x), y, z) \to \mathsf{app}(\mathsf{app}(z, \mathsf{F}_0), x)$$

$$\mathsf{f\text{-}reduce}(\mathsf{S}_2(p, q), y, z) \to \mathsf{app}(\mathsf{app}(z, \mathsf{app}(\mathsf{S}_0, p)), q)$$
$$\mathsf{f\text{-}reduce}(\mathsf{F}_2(p, q), y, z) \to \mathsf{app}(\mathsf{app}(z, \mathsf{app}(\mathsf{F}_0, p)), q)$$

Figure 1: A Complete Set of Canonical Rewrite Rules for Currified Applicative SF-calculus

Definition 4.2 (Translation of SF-calculus to $\mathsf{SF}^{\mathscr{C}}_{@}$)**.** *The translation $[\![\cdot]\!]_@$ from SF-terms to $SF^{\mathscr{C}}_{@}$-terms is defined by $[\![\mathsf{S}]\!]_@ = \mathsf{S}_0$, $[\![\mathsf{F}]\!]_@ = \mathsf{F}_0$, and $[\![MN]\!]_@ = \mathsf{app}([\![M]\!]_@, [\![N]\!]_@)$.*

We now show that $\mathsf{SF}^{\mathscr{C}}_{@}$ faithfully implements SF-calculus.

Lemma 4.3 ($[\![\cdot]\!]_@$ Preserves Reduction)**.** *Let M and N be SF-terms; if $M \to^*_{SF} N$ then $[\![M]\!]_@ \to^*_{SF^{\mathscr{C}}_{@}} [\![N]\!]_@$.*

Proof. It is sufficient to consider the basic reduction rules of SF-calculus. In the interests of clarity, we underline the redex that is contracted at each step. The case for S is straightforward:

$$\begin{aligned}[\![\mathsf{S}MNX]\!]_@ &= \mathsf{app}(\mathsf{app}(\underline{\mathsf{app}(\mathsf{S}_0, [\![M]\!]_@)}, [\![N]\!]_@), [\![X]\!]_@) \to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\underline{\mathsf{app}(\mathsf{S}_1([\![M]\!]_@), [\![N]\!]_@)}, [\![X]\!]_@) \\ &\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \underline{\mathsf{app}(\mathsf{S}_2([\![M]\!]_@, [\![N]\!]_@), [\![X]\!]_@)} \to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\mathsf{app}([\![M]\!]_@, [\![X]\!]_@), \mathsf{app}([\![N]\!]_@, [\![X]\!]_@)) \\ &= [\![MX(NX)]\!]_@\end{aligned}$$

The case for F with S the first argument (i.e. atomic) is as follows (the other atomic case is symmetric):

$$\begin{aligned}[\![\mathsf{F}\mathsf{S}MN]\!]_@ &= \mathsf{app}(\mathsf{app}(\underline{\mathsf{app}(\mathsf{F}_0, \mathsf{S}_0)}, [\![M]\!]_@), [\![N]\!]_@) \to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\underline{\mathsf{app}(\mathsf{F}_1(\mathsf{S}_0), [\![M]\!]_@)}, [\![N]\!]_@) \\ &\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \underline{\mathsf{app}(\mathsf{F}_2(\mathsf{S}_0, [\![M]\!]_@), [\![N]\!]_@)} \to_{\mathsf{SF}^{\mathscr{C}}_{@}} \underline{\mathsf{f\text{-}reduce}(\mathsf{S}_0, [\![M]\!]_@, [\![N]\!]_@)} \to_{\mathsf{SF}^{\mathscr{C}}_{@}} [\![M]\!]_@\end{aligned}$$

When the first argument to F is a factorable form, we must further consider its structure. The case for when the first argument is $\mathsf{S}X$ (for some term X) is as follows:

$$\begin{aligned}[\![\mathsf{F}(\mathsf{S}X)MN]\!]_@ &= \mathsf{app}(\mathsf{app}(\mathsf{app}(\mathsf{F}_0, \underline{\mathsf{app}(\mathsf{S}_0, [\![X]\!]_@)}), [\![M]\!]_@), [\![N]\!]_@) \\ &\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\mathsf{app}(\underline{\mathsf{app}(\mathsf{F}_0, \mathsf{S}_1([\![X]\!]_@))}, [\![M]\!]_@), [\![N]\!]_@) \\ &\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\underline{\mathsf{app}(\mathsf{F}_1(\mathsf{S}_1([\![X]\!]_@)), [\![M]\!]_@)}, [\![N]\!]_@) \to_{\mathsf{SF}^{\mathscr{C}}_{@}} \underline{\mathsf{app}(\mathsf{F}_2(\mathsf{S}_1([\![X]\!]_@), [\![M]\!]_@), [\![N]\!]_@)} \\ &\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \underline{\mathsf{f\text{-}reduce}(\mathsf{S}_1([\![X]\!]_@), [\![M]\!]_@, [\![N]\!]_@)} \to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\mathsf{app}([\![N]\!]_@, \mathsf{S}_0), [\![X]\!]_@) = [\![N\mathsf{S}X]\!]_@\end{aligned}$$

Again, the case for when the first argument is $\mathsf{F}X$ (for some term X) is symmetric and can be obtained from the above sequence by replacing each occurrence of S_0 by F_0 and each occurrence of S_1 by F_1.

The case for when the first argument is $\mathsf{S}XY$ (for some terms X and Y) is as follows:

$$[\![\mathsf{F}(\mathsf{S}XY)MN]\!]_@ = \mathsf{app}(\mathsf{app}(\mathsf{app}(\mathsf{F}_0, \mathsf{app}(\underline{\mathsf{app}(\mathsf{S}_0, [\![X]\!]_@)}, [\![Y]\!]_@)), [\![M]\!]_@), [\![N]\!]_@)$$

$$\begin{aligned}
&\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\mathsf{app}(\mathsf{app}(\mathsf{F}_0, \underline{\mathsf{app}(\mathsf{S}_1([\![X]\!]_@), [\![Y]\!]_@)}), [\![M]\!]_@), [\![N]\!]_@)\\
&\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\mathsf{app}(\underline{\mathsf{app}(\mathsf{F}_0, \mathsf{S}_2([\![X]\!]_@, [\![Y]\!]_@))}, [\![M]\!]_@), [\![N]\!]_@)\\
&\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\underline{\mathsf{app}(\mathsf{F}_1(\mathsf{S}_2([\![X]\!]_@, [\![Y]\!]_@)), [\![M]\!]_@)}, [\![N]\!]_@)\\
&\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \underline{\mathsf{app}(\mathsf{F}_2(\mathsf{S}_2([\![X]\!]_@, [\![Y]\!]_@), [\![M]\!]_@), [\![N]\!]_@)}\\
&\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \underline{\mathsf{f\text{-}reduce}(\mathsf{S}_2([\![X]\!]_@, [\![Y]\!]_@), [\![M]\!]_@, [\![N]\!]_@)}\\
&\to_{\mathsf{SF}^{\mathscr{C}}_{@}} \mathsf{app}(\mathsf{app}([\![N]\!]_@, \mathsf{app}(\mathsf{S}_0, [\![X]\!]_@)), [\![Y]\!]_@) = [\![N(\mathsf{S}X)Y]\!]_@
\end{aligned}$$

Once more, the case for when the first argument is $\mathsf{F}XY$ (for some terms X and Y) is symmetric and can be obtained from the above sequence by replacing each occurrence of S_0 by F_0, each occurrence of S_1 by F_1, and each occurrence of S_2 by F_2. □

To show that $[\![\cdot]\!]_@$ preserves strong normalisation, we will rely on the notion of a *perpetual reduction sequence*. We recall the relevant definitions of perpetual reductions and their properties [21]. A term t is called an ∞-*term* (also denoted $\infty(t)$) if it has an infinite reduction sequence. A reduction *step* $t \to s$ is called *perpetual* if $\infty(t)$ implies $\infty(s)$, that is it preserves divergence, and a reduction sequence $t_1 \to \ldots \to t_n$ is perpetual if every step $t_i \to t_{i+1}$ is perpetual. Clearly, a perpetual reduction sequence $t \to^* t'$ also preserves divergence. A *redex* u is called perpetual if its contraction in every context yields a perpetual reduction step. Let $u \to t$ be a substitution instance of a rewrite rule r (so u is a redex and t its r-contraction), then call the subterms of u that are those substituted for the variables in r the *arguments* of u. Such an argument is said to be *erased* if it corresponds to a variable that does *not* occur in the right-hand side of r. It is the case that for orthogonal rewrite systems every redex whose erased arguments are strongly normalising and closed (i.e. containing no variables) is perpetual [21, Cor. 5.1].

We first prove a couple of auxiliary lemmas.

Lemma 4.4. *Let N be an SF-normal form, then $[\![N]\!]_@$ is strongly normalising.*

Proof. We characterise the normal forms as terms taking one of the following forms: S, F, $\mathsf{S}X$, $\mathsf{F}X$, $\mathsf{S}XY$ or $\mathsf{F}XY$, in which each subterm is also a normal form. We then proceed by induction on the size of terms. The base cases, i.e. when N is either S of F are trivial since then $[\![N]\!]_@$ is itself a normal form. For the inductive cases, notice that the terms $\mathsf{S}_1([\![X]\!]_@)$, $\mathsf{F}_1([\![X]\!]_@)$, $\mathsf{S}_2([\![X]\!]_@, [\![Y]\!]_@)$ and $\mathsf{F}_2([\![X]\!]_@, [\![Y]\!]_@)$ are strongly normalising since they are head normal and by induction $[\![X]\!]_@$ and $[\![Y]\!]_@$ are strongly normalising as X and Y are by definition normal forms (smaller than N). It is then straightforward to show in each case that the (unique) reduction from $[\![N]\!]_@$ to its corresponding head normal form given above is perpetual since it does not erase any arguments, and $\mathsf{SF}^{\mathscr{C}}_{@}$ is an orthogonal rewrite system. The result then follows. □

Lemma 4.5. *Let X, M and N be SF-normal forms, and O an operator (i.e. either S or F) such that $OXMN \to_{SF} R$; then $\mathscr{C}[[\![OXMN]\!]_@] \to^*_{SF^{\mathscr{C}}_{@}} \mathscr{C}[[\![R]\!]_@]$ is a perpetual reduction sequence for any $SF^{\mathscr{C}}_{@}$-term context $\mathscr{C}$.*

Proof. We show that there is a reduction sequence $[\![OXMN]\!]_@ \to^*_{\mathsf{SF}^{\mathscr{C}}_{@}} [\![R]\!]_@$ which contracts a perpetual redex at each step, from which the result immediately follows. In fact, the reduction sequences that witness this are exactly those that are used to show preservation of reduction. In each case notice that, in the reduction sequence demonstrated in the proof of Lemma 4.3, the only erased argument (when it exists) is in the final reduction step and in each case this argument is either $[\![M]\!]_@$ or $[\![N]\!]_@$ which by Lemma 4.4 is strongly normalising since M and N are normal forms (and also closed since we do not

consider SF-terms with variables). Thus, since $\mathsf{SF}^{\mathscr{C}}_{@}$ is an orthogonal rewrite system, it follows that the redex contracted at each step is perpetual. □

We can now prove the following result.

Lemma 4.6 ($[\![\cdot]\!]_@$ Preserves Strong Normalisation)**.** *Let M be a strongly normalising SF-term; then $[\![M]\!]_@$ is strongly normalising.*

Proof. We use the same technique as used in the proof of [5, Thm. 3.4(2)], and proceed by (strong) induction on the length n of the longest reduction sequence from M to its normal form. When $n = 0$, then we have that M is a (SF-)normal form and thus $[\![M]\!]_@$ is strongly normalising w.r.t $\rightarrow^*_{\mathsf{SF}^{\mathscr{C}}_{@}}$ by Lemma 4.4. When $n > 0$ then M must contain at least one redex $OXYZ$. Consider an *innermost* redex, whose contractum is the term R. Thus $M = \mathscr{C}[OXYZ] \rightarrow_{\mathsf{SF}} \mathscr{C}[R] = N$ (for some (SF-)term context $\mathscr{C}$) and X, Y and Z are normal forms (since the redex is innermost). Now, since M is strongly normalising so too is N, and the length of its longest reduction sequence must be strictly less than n (otherwise n would not be maximum). Thus by the inductive hypothesis $[\![N]\!]_@$ is strongly normalising. Consider now the structure of $[\![M]\!]_@$: we have $[\![M]\!]_@ = \mathscr{C}'[[\![OXYZ]\!]_@]$ for some ($\mathsf{SF}^{\mathscr{C}}_{@}$-)term context $\mathscr{C}'$. Since X, Y and Z are normal forms, by Lemma 4.5 there is a perpetual reduction sequence from $[\![M]\!]_@ = \mathscr{C}'[[\![OXYZ]\!]_@]$ to $[\![N]\!]_@ = \mathscr{C}'[[\![R]\!]_@]$, i.e. one which preserves divergence. Therefore, since $[\![N]\!]_@$ is strongly normalising so too is $[\![M]\!]_@$ (if it were not, neither would $[\![N]\!]_@$ be). □

Using Berarducci and Böhm's construction, outlined in Section 3, we obtain an encoding of SF-calculus in the λ-calculus.

Definition 4.7 (Translation of SF-calculus to λ-calculus)**.** *Fix a canonical representation ϕ_{SF} of Σ_{SF} w.r.t. the rewrite rules of $\mathsf{SF}^{\mathscr{C}}_{@}$. The mapping $[\![\cdot]\!]_\lambda = (\cdot)^{\phi_{SF}} \circ [\![\cdot]\!]_@$ translates SF-calculus to λ-calculus.*

We leave it as an exercise to the reader to compute such a canonical representation ϕ_{SF}.

We now present our main result: that $[\![\cdot]\!]_\lambda$ is a faithful encoding of SF-calculus in λ-calculus.

Theorem 4.8 (Faithful Encoding of SF-calculus in λ-calculus)**.** *The translation $[\![\cdot]\!]_\lambda$ of SF-calculus into λ-calculus preserves reduction and strong normalisation.*

Proof. The result follows directly from the fact that the two translations that are composed to obtain $[\![\cdot]\!]_\lambda$, namely $[\![\cdot]\!]_@$ and $(\cdot)^{\phi_{\mathsf{SF}}}$, each satisfy both these properties. In the case of the former, we refer to Lemmas 4.3 and 4.6; for the latter to the results of Berarducci and Böhm, cf. Theorem 3.4 (note that all terms are closed, since we do not consider SF-calculus with variables). □

5 Expressiveness of Factorisation: Discussion & Related Work

We now turn out attention to the question of the expressiveness of SF-calculus relative to λ-calculus and Combinatory Logic. On one hand our results, i.e. Theorem 4.8, along with those of Jay and Given-Wilson [18], show that SF-calculus and λ-calculus simulate the same executions. On the other, SF-calculus is structure complete whereas λ-calculus is not. Is this a contradiction and, if so, how may it be resolved? Notwithstanding the long tradition of using simulations to characterise computational equivalence, it has since been realised that a refinement of this notion is necessary to draw richer, more meaningful comparisons. While a complete and in-depth analysis is not within the scope of the current paper, by discussing our results with reference to some of this work we aim to draw some concrete conclusions about the how the expressiveness of structure completeness and SF-calculus may be characterised.

The 'Standard' Notion of Equivalence. From the earliest research into computability, two aspects of abstract notions of computation were identified as relevant to the idea of expressiveness. Firstly, one wants to compare the respective set of *functions* that each model computes. For example, in recursion theory it was shown that the set of primitive recursive functions (on natural numbers) is a strict subset of the recursive functions (see e.g. [32]). At the same time, there is a requirement to compare models that operate, at a fundamental level, in diverse domains. Turing Machines, λ-calculus and Combinatory Logic are, operationally, quite different ways to compute. It was shown however that via particular (and now canonical) representations of numbers, each model can 'compute' the same set of functions on natural numbers, namely the partial recursive functions. Conversely, Gödelization allows each of the computations in these models to be *simulated* by a partial recursive function [22].

This idea of simulation extends to the operation of the models themselves: each model may simulate the operational behaviour of the others. Such simulations also appear to abstract away the problem of *representation*: often we do want to compute functions of natural numbers, however more often we desire to compute functions over different domains; it is incumbent upon us to represent elements of the desired domain of discourse as terms of the computational model. The initial characterisation of the set of 'computable' functions was over the domain of natural numbers[6], but what is the set of 'computable' functions over some other given domain? With simulations between models it seems that one may at least lay this question aside by observing that whatever can be represented in one model may then also be represented in the other, and therefore whatever functions they do compute it is the same in both cases. A stronger conclusion would be that the set of computable functions over arbitrary domains is isomorphic to the computable functions on natural numbers.

This simulation method has long since become the standard: to show two models are of equivalent computational power, demonstrate simulations of each in the other. One model of computation is only more powerful than another, then, if it is *not* possible to simulate the former in the latter. The approach has been cemented over the years, notably in Landin's now seminal work [25]. As the search space of formal computation has been explored the notion of simulation has been adapted accordingly, and there are now a number of sophisticated simulations between all sorts of models, both sequential and concurrent (e.g. [1, 29]). In this tradition, Theorem 4.8 is a result showing that λ-calculus is computationally *as powerful as* SF-calculus.

Refining the Notion of Expressiveness. It was already noted over two decades ago that despite the broad applicability and application of the simulation method, it is not actually a fine-grained enough notion to provide a complete and universal characterisation of expressiveness. Felleisen observed that since the languages we wish to compare are (usually) Turing-complete, other methods (than simulation) must be found in order to verify claims of relative (in-)expressiveness [12]. He proposed a framework based on the concept, from logic, of *eliminability* of symbols from conservative extensions [23]. One logical system $\mathscr{L}$ is a conservative extension of another system $\mathscr{L}'$ if the expressions (formulae) and theorems of the latter are subsets of those of the former. A symbol of $\mathscr{L}$ (which is not in $\mathscr{L}'$) is eliminable if there is a homomorphism (i.e. a map preserving the syntactic structure) $\varphi : \mathsf{Exp}(\mathscr{L}) \to \mathsf{Exp}(\mathscr{L}')$ from the expressions of $\mathscr{L}$ to the expressions of $\mathscr{L}'$, which acts as identity for expressions of $\mathscr{L}'$, such that an expression t is a theorem of $\mathscr{L}$ if and only if $\varphi(t)$ is a theorem of $\mathscr{L}'$. Felleisen extends this to programming languages by analogy - formulae are (syntactically valid) programs and the theorems are the terminating programs. Then we may say that language $\mathscr{L}$ is *more* expressive than language $\mathscr{L}'$ when

[6] Turing's work is different in this respect, since he deliberately embarked on a characterisation of computable functions over a different domain, namely that of strings of arbitrary symbols.

the former adds some *non-eliminable* syntactic construct, i.e. one which cannot be 'translated away'. Thus the standard simulation approach is refined by imposing an extra criterion when one language is a superset of another: can the larger one be built from the smaller one using *macros*?

To place SF-calculus in this framework we can consider SKF-calculus, i.e. the extension of SF-calculus by including an additional atomic term: the familiar K combinator. Since SKF-calculus is a proper extension of Combinatory Logic, obtained by adding the F combinator, we can apply the expressiveness test of Felleisen. That is, we ask is F eliminable? The answer to this question is *no*; indeed this is guaranteed by Theorem 2.4. In this sense, Jay and Given-Wilson's results *do* justify the claim that SF-calculus is more expressive than Combinatory Logic and λ-calculus.

More Abstract Notions of Computational Equivalence. The simulation method, and its refinement described above, are still firmly grounded in an *operational* view of computation, but recent work has sought to anchor formal comparisons of expressiveness in a more general, *abstract* basis. A notable contribution to this effort is the work of Boker and Dershowitz [7]. They abstract the notion of computational model as simply its *extension*, i.e. the set of functions over its inherent domain that it computes, and consider simulations (encodings) between them. They derive the remarkable result that combining the standard simulation approach with the natural containment of one extensionality within another leads to a paradox: some computational models can simulate models which are *strictly* more powerful in the sense that they have larger extensionalities (i.e. compute more functions). Thus, some representations *add more computational power*.

This result begs the question: do we consider simulations via such 'active' mappings to constitute an equivalence? One may suspect that the problem lies in allowing *injective* mappings between domains, and that imposing stricter conditions (e.g. bijections) would ensure 'passiveness'. This is indeed the case, but adopting such restrictions is useless for most comparisons since the passive encodings are ones which are "almost identity". We have no choice but to allow such encodings, although we do have the option of considering a hierarchy of equivalences based on the properties of the encodings used. Boker and Dershowitz define four increasingly stricter notions of (in-)equivalence based on, respectively: injective encodings (power equivalence), corresponding to the standard approach; injective encodings for which the images are computable in the simulating models (decent power equivalence); bijective encodings (bijective power equivalence); and bijections that are inverses of each other (isomorphism). Boker and Dershowitz also show that there are models (including Turing Machines and the recursive numeric functions) which cannot simulate any stronger models; they are *interpretation complete*.

We may also gain insight into the relative expressiveness of SF-calculus and λ-calculus using this framework. Our result shows that they are *power equivalent*. Theorem 2.4 shows that they cannot be bijectively power equivalent (nor, therefore, isomorphic) since that would imply that λ-calculus could distinguish arbitrary normal forms. Thus, in this sense they are not equivalent. We do not know if SF-calculus is *bijectively stronger* than λ-calculus; this would require demonstrating a bijective encoding of the latter in the former. Also, we do not know if λ-calculus is decently power equivalent to SF-calculus; we consider it at least a possibility. We would also expect that, due to its intensional capabilities, SF-calculus is interpretation complete.

Related to the work of Boker and Dershowitz, is that of Cockett and Hofstra [9], and Longley [26] which are both concerned with category-theoretic descriptions of abstract computational models. In these frameworks model equivalence is interpreted by categorical isomorphism, and so akin to the strongest notion of equivalence considered by Boker and Dershowitz.

6 Conclusions & Future Work

In this paper, we have considered the relationship of the recently introduced SF-calculus to the 'canonical' computational model of λ-calculus and, so by extension, Combinatory Logic. We have demonstrated that SF-calculus can be faithfully encoded (i.e. simulated) in the λ-calculus by defining a behaviourally equivalent applicative term rewriting system and then interpreting this system in λ-calculus using a construction of Berarducci and Böhm. This result shows that SF-calculus and λ-calculus are of equivalent computational power, according to the classical interpretation of computational equivalence. We have also considered the relationship of SF-calculus to the λ-calculus using a more nuanced interpretation of equivalence, informed by research in the literature. Moreover, we hope to have exposed both SF-calculus and Berarducci and Böhm's encoding to greater prominence. We feel that they are both subjects of great interest which deserve to be better known.

With respect to future work, there is still great scope for investigating the expressiveness of SF-calculus. There are the open questions we have highlighted regarding SF-calculus as it relates to the framework of Boker and Dershowitz. The categorical and denotational natures of SF-calculus also deserve exploration. Beyond this, the wider question of how best to qualify computational expressiveness still remains; we believe the study of SF-calculus can provide further insights on this. For example, the structural completeness property is already a new metric; Jay and Vergara have also considered a strengthening of the notion of decent power equivalence in which the simulation of each model in itself via the composition of the two encodings is computable in each model [19]. Quantitative questions regarding expressiveness also present themselves: e.g. our encoding of SF-calculus in λ-calculus leads to a large increase in the size of terms; are there lower bounds on the size of such increases which meaningfully quantify expressive power?

Acknowledgements We would like to thank Barry Jay, and also the anonymous reviewers for helpful comments in the preparation of the final version of this paper. This research was supported by EPSRC Grant EP/K040049/1.

References

[1] M. Abadi & L. Cardelli (1996): *A Theory Of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[2] Martín Abadi, Luca Cardelli & Ramesh Viswanathan (1996): *An Interpretation of Objects and Object Types*. In: *Principles of Programming Languages (POPL'96)*, pp. 396–409, doi:10.1145/237721.237809.

[3] S. van Bakel & M. Fernández (1997): *Normalization Results for Typeable Rewrite Systems. Information and Computation* 133(2), pp. 73–116, doi:10.1006/inco.1996.2617.

[4] H. Barendregt (1981): *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam.

[5] Alessandro Berarducci & Corrado Böhm (1992): *A Self-Interpreter of Lambda Calculus Having a Normal Form*. In: *Computer Science Logic, 6th Workshop, CSL '92*, pp. 85–99, doi:10.1007/3-540-56992-8_7.

[6] Corrado Böhm, Adolfo Piperno & Stefano Guerrini (1994): *Lambda-Definition of Function(al)s by Normal Forms*. In: *ESOP'94, Edinburgh, U.K.*, pp. 135–149, doi:10.1007/3-540-57880-3_9.

[7] Udi Boker & Nachum Dershowitz (2009): *The Influence of Domain Interpretations on Computational Models. Applied Mathematics and Computation* 215(4), pp. 1323–1339, doi:10.1016/j.amc.2009.04.063.

[8] Alonzo Church (1936): *An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics* 58(2), pp. 345–363, doi:10.2307/2371045.

[9] J.R.B. Cockett & Pieter J.W. Hofstra (2010): *Categorical Simulations*. *Journal of Pure and Applied Algebra* 214(10), pp. 1835 – 1853, doi:10.1016/j.jpaa.2009.12.028.

[10] H. B. Curry & R. Feys (1958): *Combinatory Logic*. 1, North-Holland, Amsterdam.

[11] H. B. Curry, J. R. Hindley & J. P. Seldin (1972): *Combinatory Logic*. 2, North-Holland, Amsterdam.

[12] Matthias Felleisen (1991): *On the Expressive Power of Programming Languages*. *Sci. Comput. Program.* 17(1-3), pp. 35–75, doi:10.1016/0167-6423(91)90036-W.

[13] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (1995): *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[14] Thomas Given-Wilson (2014): *Expressiveness via Intensionality and Concurrency*. In: *Theoretical Aspects of Computing - ICTAC 2014, Bucharest, Romania*, pp. 206–223, doi:10.1007/978-3-319-10882-7_13.

[15] Daniele Gorla (2010): *Towards a Unified Approach to Encodability and Separation Results for Process Calculi*. *Inf. Comput.* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[16] J.M. Jansen, P. W. M. Koopman & R. Plasmeijer (2006): *Efficient Interpretation by Transforming Data Types and Patterns to Functions*. In: *Trends in Functional Programming, Nottingham, UK*, pp. 73–90.

[17] Barry Jay (2009): *Pattern Calculus - Computing with Functions and Structures*. Springer.

[18] Barry Jay & Thomas Given-Wilson (2011): *A Combinatory Account of Internal Structure*. *J. Symb. Log.* 76(3), pp. 807–826, doi:10.2178/jsl/1309952521.

[19] Barry Jay & Jose Vergara (2014): *Confusion in the Church-Turing Thesis*. Available at http://arxiv.org/abs/1410.7103. CoRR abs/1410.7103.

[20] Stefan Kahrs (1995): *Confluence of Curried Term-Rewriting Systems*. *J. Symb. Comput.* 19(6), pp. 601–623, doi:10.1006/jsco.1995.1035.

[21] Zurab Khasidashvili, Mizuhito Ogawa & Vincent van Oostrom (2001): *Perpetuality and Uniform Normalization in Orthogonal Rewrite Systems*. *Inf. Comput.* 164(1), pp. 118–151, doi:10.1006/inco.2000.2888.

[22] S. C. Kleene (1936): *Lambda-definability and recursiveness*. *Duke Mathematical Journal* 2, pp. 340–353, doi:10.1215/S0012-7094-36-00227-2.

[23] S.C. Kleene (1952): *Introduction to Metamathematics*. Bibliotheca Mathematica, Wolters-Noordhoff.

[24] J. W. Klop (1992): *Term Rewriting Systems*. In S. Abramsky, Dov M. Gabbay & S. E. Maibaum, editors: *Handbook of Logic in Computer Science (Vol. 2)*, OUP, Inc., New York, NY, USA, pp. 1–116.

[25] Peter J. Landin (1966): *The Next 700 Programming Languages*. *Commun. ACM* 9(3), pp. 157–166, doi:10.1145/365230.365257.

[26] John Longley (2014): *Computability Structures, Simulations and Realizability*. *Mathematical Structures in Computer Science* 24(2), doi:10.1017/S0960129513000182.

[27] Torben Æ. Mogensen (1992): *Efficient Self-Interpretation in Lambda Calculus*. *Journal of Functional Programming* 2, pp. 345–364, doi:10.1017/S0956796800000423.

[28] Jens Palsberg & C. Barry Jay (1998): *The Essence of the Visitor Pattern*. In: *COMPSAC '98, August 19-21, 1998, Vienna, Austria*, pp. 9–15, doi:10.1109/CMPSAC.1998.716629.

[29] D. Sangiorgi & D. Walker (2001): *PI-Calculus: A Theory of Mobile Processes*. CUP, New York, NY, USA.

[30] J. Steensgaard-Madsen (1989): *Typed Representation of Objects by Functions*. *ACM Transactions on Programming Languages and Systems* 11(1), pp. 67–89, doi:10.1145/59287.77345.

[31] Aaron Stump (2009): *Directly Reflective Meta-Programming*. *Higher-Order and Symbolic Computation* 22(2), pp. 115–144, doi:10.1007/s10990-007-9022-0.

[32] G.J. Tourlakis (1984): *Computability*. Reston Publishing Company.

[33] A. M. Turing (1937): *On Computable Numbers, with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* s2-42(1), pp. 230–265, doi:10.1112/plms/s2-42.1.230.